효율적인 이벤트 큐의 구조에 관한 연구

A Study on the Structures for Efficient Event Queues

김 상 욱*, 유 승 헌**, 정 연 모**
Sang-uk Kim, Seung-heun Yoo, Yunmo Chung

Abstract

The performance of event-driven logic simulation frequently used for VLSI design verification depends on the data structures for event queues. This paper improves the existing Timing Wheel as a data structure for an event queue. In case of the use of B+ tree, an efficient node degree is also presented based on the experiment results. A new Timing Wheel index structure, which eliminates the insertion and deletion overhead of B+ tree, is proposed and analyzed.

1. 서 론

VLSI 설계 과정에서 오류 및 논리 검증을 위한 시뮬레이션은 필수적이지만 하드웨어의 집적도 및 복잡도가 증가함에 따라서 이를 위한 소요시간이 많이 걸린다는 것은 잘 알려져있는 사실이다. 현재 널리 사용되고 있는 시뮬레이션의 종류로는 논리 레벨 설계에서 사용되고 있는 이벤트-구동 논리 시뮬레이션(event-driven logic simulation)이다. 이런 이벤트-구동 방식은 각 게이트에서 시뮬레이션 시간과 이진 신호, 소속된 게이트 번호 등을 가지고 있는 이벤트를 서로 주고 받음으로서 이벤트가 있는 게이트만을 수행하는 선택적 추적 접근(selective trace approach) 방법을 사용하며 다중 지연 논리 회로 시뮬레이션을 위해서 사용될 수도 있다[5]. 이러한 이벤트-구동 시뮬레이션은 이벤트 큐에 저장된 이벤트를 사용하여 수행한 결과로 인해 생긴 새로운 이벤트를 이벤트 큐에 저장하는 과정을 반복함으로서 시뮬레이션을 할 수 있다. 그러므로 이벤트-구동 시뮬레이션에서의 이벤트 큐의 효율적인 유지 및 관리는 시뮬레이션의 좋은 성능을 위해서 중요한 부분 중의 하나이다.

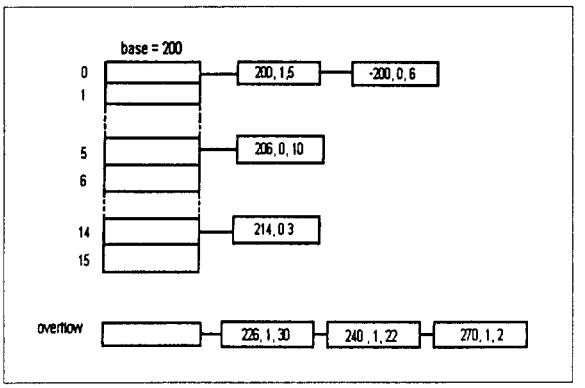
본 논문에서는 이벤트-구동 시뮬레이션 과정에서의 이벤트 큐의 유지와 또한 이벤트의 저장 및 검색에서 가장 시간이 많이 걸리는 것으로 알려진 이벤트 큐를 위한 자료구조에 대해서 연구하고자 한다. 먼저 지금까지 효율적인 것으로 알려져 있는 이벤트 큐의 자료구조로서 널리 사용되어온 해싱(hashing)을 이용한 기존의 Timing Wheel의 기능을 개선하였다. 또한 보다 빠르게 이벤트를 저장 및 검색할 수 있도록 B+ 트리를 자료구조로 응용한 경우에 이벤트 큐의 우수한 성능을 위한 가장 적절한 노드 차수를 실험적으로 제시하였다. 그리고 B+ 트리에서의

* 한국과학기술연구원 정보전자연구부
** 경희대학교 전자공학과

삽입과 삭제의 오버헤드를 없앤 새로운 자료구조를 제시하고 수행 능력을 비교하며 실제 시뮬레이션에서 사용 가능성을 연구하였다.

2. 기존의 Timing Wheel 구조

이벤트 큐의 유지 및 검색에서 효율적인 기법으로써 해싱에 기초를 둔 Timing Wheel을 다양하게 사용되고 있음을 알 수 있다. 이러한 방법은 RSIM에서 Terman에 의해 사용되었고 LDVSIM에서 사용되고 있다[5]. 이러한 Timing Wheel은 〈그림 1〉에서 처럼 분리된 key들에서 동작하는 일종의 priority 큐 구조이며 저장될 이벤트는 시뮬레이션 시간이 key로 사용된다[1,2].



〈그림 1〉 기존의 Timing Wheel 구조

Wheel은 유한한 크기(hashsize) D와 base B를 가지고 있으며 B보다 작은 시뮬레이션 시간을 가진 이벤트는 허용되지 않는다. 즉 이벤트들 중의 최소 시뮬레이션 시간은 결코 감소하지 않는 값들이다.

시뮬레이션 과정에서 생성되는 이벤트가 B+D보다 적은 시간을 가지면 wheel내의 해당 bucket에 삽입된다. 여기서 bucket이란 같은 시뮬레이션 시간을 가진 이벤트를 링크로 연결하여 모아 놓은 리스트를 말한다. 만약 B+D보다 크면 그것은 증가순으로 overflow에 추가된다. 그리고 이벤트의 출력은 wheel의 첫번째의 bucket에 있는 이벤트부터 시작된다. 첫 bucket에 있는 모든 이벤트가 제거될때 base B는 다음 bucket으로 이동되고 그 bucket 내의 이벤트가 제거된다. Wheel에 있는 이벤트가 다 사용되면 overflow에 있는 이벤트들 중에서 wheel의 크기만큼의 이벤트만 골라 wheel에 추가한 뒤 사용한다[5,6].

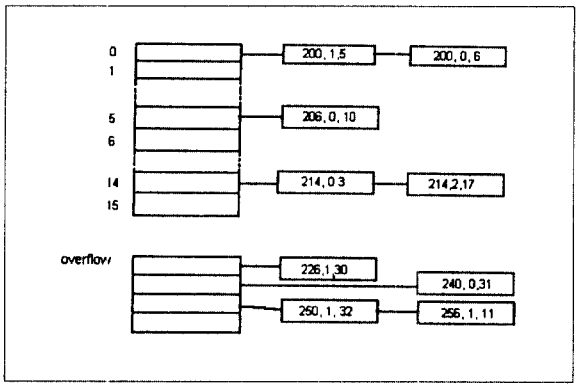
〈그림 1〉에서 hashsize는 16이고 base는 200이므로 200부터 215 사이에 있는 요소는 wheel에 저장되고 이보다 더 큰 시뮬레이션 시간을 가진 이벤트는 overflow 리스트에 삽입된다. 여기서 이벤트의 시뮬레이션 시간이 B+D보다 작다면 일정한 시간 안에 추가된다. 그러나 wheel의 크기가 너무 작으면 이벤트들은 O(n)시간 동안에 overflow list에 추가될 것이다(여기서 n은 overflow 리스트의 길이). 만약 Timing wheel이 너무 크면 Timing wheel을 진행시키는데 O(D) time이 걸릴 것이다.

3. Overflow 리스트를 개선한 Timing Wheel

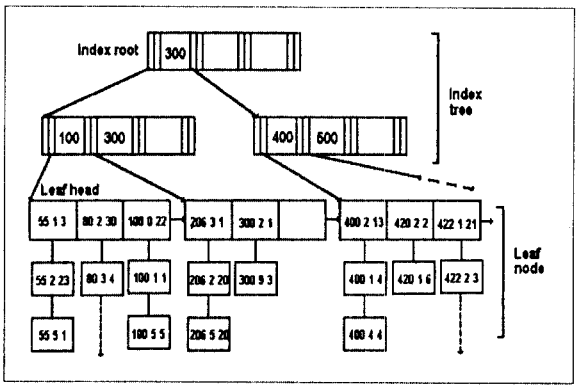
위 〈그림 1〉에서 보여준 overfolw 리스트를 가진 Timing Wheel에서는 overflow에 들어갈 이벤트의 시뮬레이션 시간의 범위가 적으면 효율적이다. 그러나 wheel의 크기가 작거나 생성되는 이벤트의 범위가 넓으면 overflow 리스트에 넣어야 할 이벤트의 수가 많아지므로 그 overhead는 아주 크다. 그러나 wheel의 크기를 미리 넓게 잡아 주면 되지만 이는 예측하기가 힘들다. 또한 하나의 overflow 리스트에 들어가는 모든 이벤트를 시뮬레이션 시간 증가순으로 정렬을 유지하기란 쉽지가 않다. 그러므로 〈그림 2〉에서처럼 overflow 리스트를 여러 개 만들어 이들끼리 연결하도록 하며 하나의 overflow 리스트 내에서는 이벤트들이 정렬되어 유지될 필요가 없다. 하나의 overflow 리스트는 Wheel의 크기만큼의 범위를 가진 이벤트를 가진다. 시뮬레이션시 Wheel을 다 사용하면 overflow 리스트에서 다시 이벤트들을 가져와 사용한다.

회로의 시뮬레이션 진행시 wheel의 범위를 벗어난 이벤트는 overflow 리스트중의 하나에 저장된다. 따라서 wheel의 범위가 작아도 여러 개의 overflow 리스트로 인하여 마지막 overflow 리스트에 저장되는 이벤트의 수가 작아지므로 그 overhead가 작아진다.

〈그림 2〉에서 hashsize는 16이며 4개의 overflow 리스트가 있다. 이 overflow 리스트의 수를 overflowsize라고 부르며 위에서는 4개의 overflow 리스트를 가지고 있으므로 overflowsize는 4이다. 마지막을 제외한 나머지의 경우에는 하나의 overflow 리스트가 가지는 이벤트의 범위는 hashsize 만큼의 크기를 계속 가진다. 각 overflow 리스트에 들어가



〈그림 2〉 Overflow 리스트를 개선한 Timing Wheel



〈그림 3〉 B+ 구조를 이용한 이벤트 큐

는 이벤트는 정렬할 피요가 없다. 그러므로 첫 overflow 리스트는 이벤트의 시뮬레이션 시간이 216에서 231까지, 두 번째 리스트는 232에서 247까지, … 을 가진다. 앞의 세 overflow 리스트에 넣을 수 없는 이벤트는 마지막 리스트에 들어가며 처음 리스트가 Wheel로 옮겨지면 마지막 리스트를 제외한 각 리스트는 그 시작 주소를 앞 리스트에 옮긴다. 세 번째 리스트는 마지막 리스트에 있는 이벤트들 중에서 해당하는 것을 골라 연결한다.

이 구조를 MOTW(Modified Overflowlist Timing Wheel)라 하자.

4. B+ 트리 구조를 이용한 이벤트 큐

시뮬레이션시에 이벤트의 갯수가 많아지면 이벤트 큐를 통한 시뮬레이션 수행 속도가 늦어지므로 이를 개선하기 위해서 인덱스(index) 구조를 사용하여 빠르게 시뮬레이션을 수행할 수가 있다. 즉 인덱스 구조를 이용한 구조중에서 대표적인 것이 B+ 트리 구조이다[3,4].

〈그림 3〉은 B+ 트리를 이용한 이벤트 큐를 나타낸 것이다. 위의 이벤트 큐는 leaf 노드를 제외한 인덱스 부분과 leaf 노드로 구성된 순차 데이타 부분(sequence data set)으로 나뉘어져 있다. 인덱스의 key값은 이벤트의 삽입시 사용된다. 즉 해당 이벤트의 시뮬레이션 시간을 가진 leaf 노드를 신속하게 직접 찾아가기 위한 수단으로만 사용된다. Root 노드를 제외한 모든 노드의 차수는 $\lceil M/2 \rceil$ 이상이어야 한다. 〈그림 3〉에서는 차수가 3 이므로 각 노드는 적어도 2개의 key값을 가져야 한다. 여기서 차수란 노드의 크기로서 한 노드에 들어갈 수 있는 key의 갯수를 말한다.

시뮬레이션을 위한 모든 이벤트는 leaf 노드에서 순차적으로 나열되고 이들끼리 차례로 연결된다. 연결된 leaf 노드는 기존의 Timing Wheel 구조에서 overflow를 제거한 wheel만을 가지므로 overflow를 고려할 필요가 없다. 따라서 leaf 노드는 이벤트 큐로서 시뮬레이션을 위해 곧바로 사용할 수 있다. 여기서 인덱스의 최대 높이는 $\log_{\lceil M/2 \rceil}(N+1)/2$ 이므로 이벤트 큐로서의 입출력 속도는 차수 M의 B+ 트리에서는 $O(\log_M N)$의 성능을 가진다. 여기서 N은 전체 키의 개수를 말한다.

〈그림 3〉의 B+ 트리를 이용한 이벤트 큐에서 삽입 및 출력 알고리듬은 다음과 같다.

삽입 알고리듬

(1) 생성된 새로운 이벤트의 삽입은 인덱스의 root에서부터 key값을 비교하여 이벤트 큐의 해당 leaf 노드에 추가되며 해당 노드가 없는 경우에는 새로 생성하여 삽입한다.

(2) 해당 leaf 노드에 key가 다 차 있지 않는 경우에는 정렬하여 삽입한다.

(3) Key값이 같을 경우에는 leaf 노드의 해당 bucket에 삽입한다.

(4) Leaf 노드에 key값이 다 차 있는 경우에는 입력할 key를 그 노드의 key값들 사이에 정렬하여 $\lceil M/2 \rceil$ 번째 key값을 인덱스로 올리며 새로운 leaf 노드를 동적으로 생성하여 연결해 준다.

(5) Leaf 노드의 ⌈M/2⌉번째보다 큰 key들은 새로 생성된 leaf 노드에 삽입한다.

출력 알고리듬

(1) 이벤트 큐의 처음 leaf 노드에서 첫 bucket의 key값을 출력하며 차례대로 진행한다.

(2) 하나의 bucket을 다 사용한 뒤에 블럭의 key의 갯수가 ⌈M/2⌉이면 연결된 다음 leaf 노드의 첫 bucket에 있는 key를 가져와 재배치한다.

(3) 만약 첫 bucket을 앞 노드에 보낸 다음 노드의 key의 갯수가 ⌈M/2⌉ 미만이면 첫 leaf 노드와 합병한다. 또한 연결된 leaf 노드는 삭제되며 인덱스 부분에서의 key값도 삭제한다.

(4) 시뮬레이션이 수행하고 있는 동안은 이벤트 큐의 첫 leaf 노드는 삭제되지 않으며 완료시 삭제된다.

이 구조를 B+TW(B+ tree Timing Wheel)이라 하자.

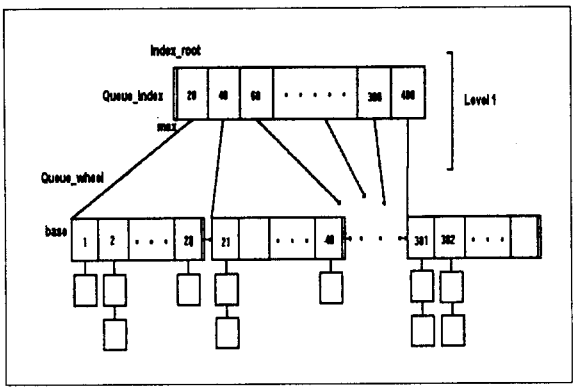
일반적으로 시뮬레이션시에 이벤트의 분포를 보면 초기에는 이벤트가 주로 생성되지만 어느정도 시간이 지나면 삽입과 출력되는 이벤트의 수가 비슷하므로 이벤트 큐의 레벨의 깊이는 거의 일정하게 유지된다.

5. Timing Wheel 구조의 개선

B+ 트리를 이용한 이벤트 큐 구조는 입력시에 노드의 키가 다 차면 분열을 하여야 하며 또한 노드의 개수가⌈N/2⌉ 미만인 경우에 재배치 및 합병을 해야 하므로 시간이 다시 요구되는 단점이 있다. 이러한 단점을 해결하기 위하여 본 논문에서는 인덱스 구조와 연결리스트로 이루어져 있으며 해싱을 적용한 새로운 이벤트 큐 구조를 제시하고자 한다. 그 구조가 〈그림 4〉에 나타나 있다.

개선한 Timing Wheel 구조는 B+ 트리 구조와 같이 queue_index 와 queue_wheel의 두 부분으로 나뉘어져 있으며 queue_index는 queue_wheel 까지의 경로를 제공하고 queue_wheel은 leaf 노드들의 연결 리스트로서 이벤트들을 가지고 있다.

B+ 트리와의 차이점을 설명하기로 한다. B+ 트리에서는 각 노드가 보유해야 할 key의 개수에 제한이 있지만 개선한 Timing Wheel에서는 이러한 제한이 없으며 각 노드에 들어가야할 key들의 값과 위치가 정해진다. leaf 노드에서 모든 시뮬레이션 시간을 가지도록 한다. 그러므로 B+ 트리와 같이 분열, 재배치, 합병이 필요없다. 실제 시뮬레이션시에 이벤트가 거의 모든 시뮬레이션 시간에 걸쳐 생성되는 분포를 가지므로 이 구조를 이벤트 큐로서 효율적으로 사용할 수 있다. 특히 회로가 크거나 복잡한 경우에는 이벤트 수가 많아지므로 이 구조가 적합하다.



〈그림 4〉 개선한 Timing Wheel

〈그림 4〉에서 차수가 20이므로 Queue_wheel의 각 leaf 노드들은 hashsize가 20이다. 따라서 시뮬레이션 시작시 첫 leaf 노드에는 1에서 20까지의 이벤트가 입력이 되며 다음 노드에서는 21에서 40까지의 이벤트가 입력이 되며 이와 같이 계속된다. 그러나 이벤트가 없는 경우에는 leaf노드가 생성되지 않는다. 만약 시뮬레이션 시간이 기존의 이벤트 큐의 범위를 초과한 경우에는 새로운 root가 생성되며 기존의 root가 새로운 root의 child로 들어간다. 예를 들어 〈그림 4〉에서 시뮬레이션 시간이 420을 가진 이벤트가 발생한 경우에는 〈그림 5〉와 같이 새로운 root가 생성된다. 이때 420의 이벤트가 삽입될 leaf 노드가 생성되며 이 노드의 시작 위치를 저장할 인덱스의 노드도 또한 생성되어 각각의 시작위치가 인덱스에 저장된다.

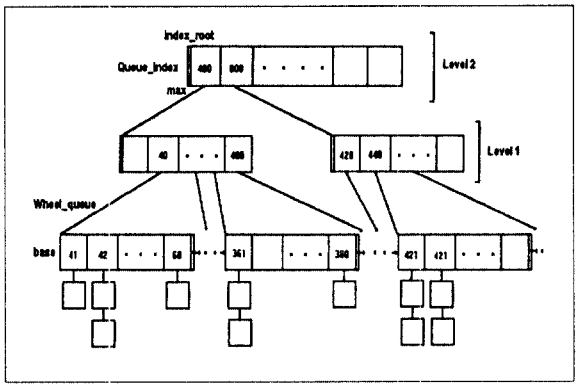
본 논문에서의 제시된 구조의 해싱함수, 삽입, 출력 알고리듬은 다음과 같다.

해싱 함수

(1) Index_root에서의 해싱함수 :

m_root = ⌊(이벤트 key값)/(pow(M,level의 수))⌋

m_root_mode = (이벤트 key값)%(pow(M,level의 수))



〈그림 5〉 인덱스가 분열된 개선한 Timing Wheel

```
if (m_root_mode == 0)
  { m_root = m_root-1 }
```

(2) Index_root를 제외한 인덱스에서 leaf 노드까지의 해싱함수 :

```
m_index = ⌊((이벤트 key값)/(pow(M,level))⌋
- ((부모레벨내의 위치)*M)
m_index_mode = (이벤트 key값)%(pow(M,level의 수))
if (m_index_mode == 0)
  { m_index = m_index-1 }
```

(3) Leaf 노드에서의 해싱 함수 :

```
m_leaf = (이벤트 key값)%M
if (m_leaf == 0)
  { m_leaf = M-1 }
else
  { m_leaf = m_leaf - 1 }
```

삽입 알고리듬

(1) 생성된 새로운 이벤트의 삽입은 인덱스의 index_root로부터 해싱을 하여 Wheel_queue의 해당 leaf 노드에 삽입된다. (해싱함수 (1), (2), (3) 이용)

(2) 삽입시 wheel_queue의 해당 leaf 노드가 생성되어 있지 않으면 생성하여 Queue_index에 생성된 노드의 시작 위치를 저장한다. 또한 Wheel_queue에도 서로 연결한다.

(3) Leaf 노드에서 key값이 같을 경우에는 해당 bucket에 삽입한다.

(4) 생성된 이벤트의 키값이 Queue_index의 범위(max)를 벗어나면 인덱스를 분열하여 새로운 인덱스구조를 만들어 연결하여 삽입한다.(〈그림 4〉 ⇒ 〈그림 5〉, 해싱함수 (1), (2), (3) 사용)

출력 알고리듬

(1) Queue_wheel의 첫 leaf 노드에서 첫 bucket의 이벤트를 출력하면서 시뮬레이션이 진행한다.

(2) 출력이 끝난 leaf 노드는 삭제하며 삭제된 노드의 시작 위치를 저장하고 있는 인덱스내의 저장된 주소값도 삭제한다.

(3) Queue_index내의 인덱스가 포함하고 있는 queue_wheel의 블록이 다 사용하면 그 인덱스도 또한 삭제하면서 부모 index내의 삭제할 index의 시작위치의 주소값도 삭제한다.

(4) Queue_wheel내의 모든 이벤트가 다 사용되면 시뮬레이션은 종료된다.

이 구조를 MTW(Modefied Timing Wheel)이라 하자.

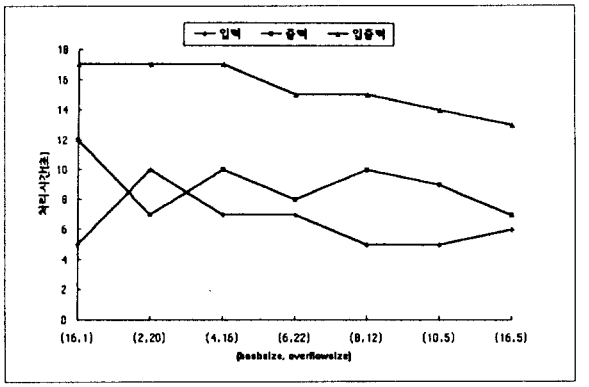
6. 실험 결과 및 평가

각각의 이벤트 큐의 성능을 서로 비교하기 위해서 승산기(multiplier)에서 생성되는 이벤트들을 사용하여 이벤트 큐의 성능을 비교하였다.

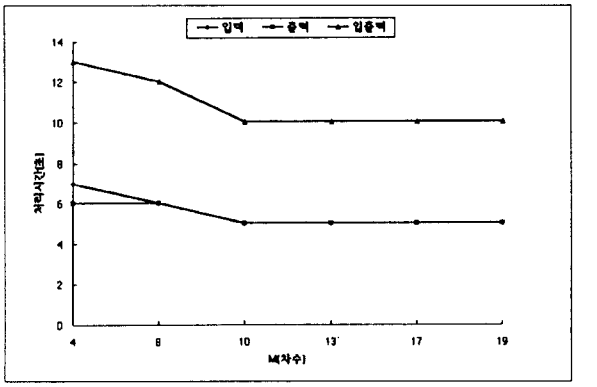
먼저 overflow 리스트를 개선한 구조에서 다중 지연을 고려한 이벤트 큐에 대해 살펴보자.

〈그림 6〉에서 hashsize가 16이고 overflowsize가 1인 경우는 기존의 Timing Wheel의 구조를 나타낸다. 그리고 hashsize와 overflowsize에 따라 수행속도가 달라짐을 알 수 있다. 즉 hashsize가 커짐에 따라 수행 속도가 빨라짐을 보이고 있다. 〈그림 6〉에서는 hashsize가 16이고 overflowsize가 5인 경우가 가장 빠른 수행 시간을 나타내고 있다. Hashsize가 작고 overflowsize가 큰 경우는 overflow 리스트에 있는 이벤트들을 자주 wheel로 옮겨야 하므로 수행 능력이 떨어진다. 따라서 overflow 리스트에 삽입되는 이벤트보다 wheel에 삽입되는 이벤트가 많을수록 수행 속도가 빨라진다.

〈그림 7〉에서는 B+ 트리를 사용한 이벤트 큐의 차수에 따른 처리시간을 나타내고 있다. 일반적으로 트리의 레

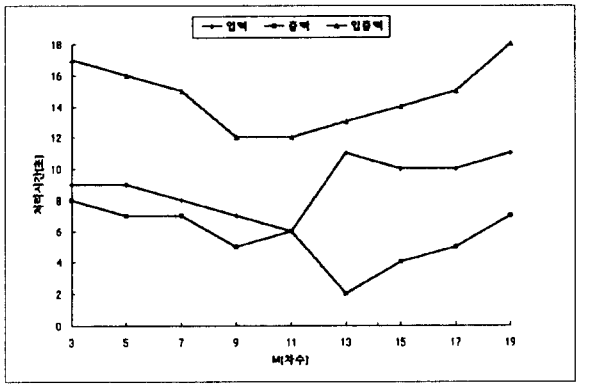


〈그림 6〉 Overflow 리스트를 개선한 구조에서의 수행 속도



〈그림 8〉 Wheel—index의 크기에 따른 수행 속도

벨의 수 즉 깊이가 늘어나면 삽입을 위한 시간이 소요되므로 차수를 높여 레벨을 최소화 하여야 한다. 〈그림 7〉에서 보면 차수가 11이 될 때 까지는 처리속도가 감소한다. 그러나 그 이후에는 다시 처리속도가 증가함을 볼 수 있다. 그 이유는 노드의 크기가 커지므로 분열, 합병, 재배치를 위한 오버헤드가 커지기 때문이다.



〈그림 7〉 차수에 따른 B+ 트리 큐의 처리시간 비교

인덱스를 사용하여 개선된 Timing Wheel에서의 차수의 크기에 따른 수행 속도가 그림 8에 보여지고 있다. 삽입시 차수의 크기에 따라 레벨의 깊이가 달라지므로 해싱의 횟수가 달라진다. 따라서 차수가 10이 될 때까지는 해싱의 횟수로 인하여 시간의 차이가 난다. 그러나 차수가 10 이상이 되면 일정한 시간내에 삽입이 된다. 출력은 leaf 노드에서 바로 출력이 되므로 차수가 10이상인 경우에는 해싱으로 인하여 같은 시간내에 입출력이 수행된다.

Wheel_index의 차수는 leaf 노드의 hashsize와 같다. 따라서 Wheel_index에서 한 노드가 포함할 이벤트 범위는 〈그림 5〉에서 보면 $(hashsize)^{(level+1)}$이다. 시뮬레이션 과정에서 회로의 이벤트 생성 범위나 이벤트 분포가 크더라도 기존의 Timing Wheel의 hashsize나 크지 않은 차수로도 적은 레벨에서 수행할 수 있다.

기존의 Timing Wheel과 overflow 리스트를 개선한 Timing Wheel, B+ 트리를 이용한 구조, 개선된 이벤트 큐 구조의 수행 속도를 단일 큐의 수행 속도와 비교해보자. 먼저 단일 큐는 오직 하나의 리스트만을 가진 이벤트 큐이다. 이벤트의 삽입은 큐의 제일 처음 부분의 이벤트부터 비교하여 정렬하면서 삽입이 된다. 출력은 앞에서 순서대로 이벤트가 출력된다. 지금까지 간단한 시뮬레이션의 구현을 위해서 단일 큐가 사용되며 시뮬레이션을 위한 큐의 구현 및 성능 평가를 위한 기본 구조로서 단일 큐 구조를 사용한다. 따라서 단일 큐 구조와의 수행 능력 비교가 〈표 1〉에 나타나 있다. 기존의 Timing Wheel에서 hashsize는 overflow 리스트를 개선한 Timing Wheel과 같은 크기인 16으로 하였으며 overflow 리스트를 개선한 Timing Wheel에서 hashsize가 16이고 overflowsize가 5인 구조를 사용하였다. B+ 트리 구조의 이벤트 큐는 차수가 11인 경우 사용하였는데 이는 차수가 9인 경우보다 입력과 출력의 처리시간 간격이 적기 때문이다. 그리고 개선한 이벤트 큐 구조는 차수를 10으로 하였다. 여기서의 실험 결과는 단일 지연과 다중 지연을 나누어서 비교를 하였다. 그리고 그 수행 결과를 입력, 출력, 입출력 시간으로 비교하였다.

〈표 1〉 수행 능력의 비교

(단위 : 초)

	단일지연			다중지연		
	입력	출력	입출력	입력	출력	입출력
단일 큐	1058	4	1062	968	4	972
TW	5	12	17	5	12	17
MOTW	5	7	12	6	7	13
B+EQ	8	5	13	6	6	12
MTW	5	5	10	5	5	10

〈표 1〉에서 단일 큐를 사용한 경우는 시뮬레이션 과정에서 생성된 새로운 이벤트의 삽입시 정렬을 해야 하므로 가장 많은 시간을 요구하는 반면에 출력은 빠른 시간내에 수행을 한다. TW는 MOTW에 비해 출력시 단일 overflow 리스트에서 wheel로 이벤트들을 삽입시 시간이 소비되는 것을 보이고 있다. 단일 지연과 다중지연에서 MTW는 이벤트의 위치가 정해져 있으므로 해싱을 이용하였기에 입출력이 같은 시간을 보이고 있으며 다른 구조에 비해 빠른 수행 시간을 보이고 있다. 단일 큐에서의 수행 시간을 MTW에서의 수행 시간으로 나눈 속도비(speed rate)를 보면 다중 지연에서 97.2의 속도 향상을 보였다.

실험 결과 MTW는 입출력이 다른 구조에 비해서 속도 향상이 있음을 알 수 있다. 이는 가장 많이 쓰이는 기존의 Timing Wheel보다 본 논문에서 제시한 구조를 이용한 이벤트 큐 구조가 더 효율적인 것을 보이고 있다. 이벤트-구동 시뮬레이션에서 이벤트 큐의 자료구조로서 이 구조를 이용하여 시뮬레이션을 할 경우 보다 빠른 시간 내에 시뮬레이션을 가능하리라 본다.

7. 결론

VLSI 설계시 시간이 많이 걸리는 시뮬레이션 과정 중에서도 이벤트 큐에 이벤트를 저장하고 검색하는 데 시간이 가장 많이 걸린다는 것은 누구나 잘 알고 있다. 따라서 본 논문에서는 이를 개선하기 위하여 기존의 해싱을 이용한 이벤트 큐에서 Timing Wheel의 자료구조를 개선하였다. 또한 인덱스 구조를 이용한 B+ 트리 구조와 이 구조의 문제점을 개선한 새로운 Timing Wheel 구조를 제시하였으며 기존의 자료 구조와의 성능을 실험 비교하였다. 실험 결과에 따르면 본 논문에서 제시한 구조가 가장 효율적임을 알 수 있었으며 이러한 구조를 이용하여 시뮬레이터의 이벤트 큐로써 사용할 수 있다고 본다.

참고문헌

[1] Alexander Miczo, "Digital Logic Testing and Simulaltion", 1986.

[2] Breuer & Friedman, "Diagnosis & Reliable Design of Digital Systems", computer scince press, 1986.

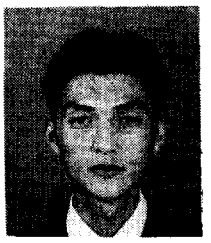
[3] Douglas Comer, "The Ubiquitous B-Tree", Computing Surveys, vol 11, No2, June 1979, pp121-pp137

[4] Horowitz Sahni Anderson-Freed, "Fundamentals of Data Structures in C", computer sinence press, 1993.

[5] Jack & Vedder & Briner. Jr, "Parallel Mixed-Level Simulation of Digital Circuits Using Virtual Time", Technical Report TR90-38, 1990.

[6] Stephen A. Szygenda & Cliff W. Hemming & John M. Hemphill, "Time Flow Mechanisms for Use in Digital Logic Simulation", Computr Scince/Operations research Center, Sourthern Methodist University.

● 저자소개 ●



김상욱
1993년 2월 경희대학교 졸업
1995년 2월 경희대학교 대학원 졸업(석사)
1995년 3월~현재 한국과학기술연구원 정보전자연구부 연구원

유승헌
1994년 2월 경희대학교 졸업
1996년 2월 경희대학교 대학원 졸업 예정(석사)
1996년 1월 현대전자 입사 예정

정연모
1980년 경북대학교 졸업(학사)
1982년 KAIST 졸업(공학석사)
1982년~1987년 경제기획원 전산처리관
1992년 미국 미시간주립대학교 졸업(공학박사)
1992년~현재 경희대학교 전자공학과 조교수
관심분야 : VLSI설계 및 CAD, 병렬처리 시뮬레이션, 컴퓨터구조 등